12 Dynamic client registration

This chapter covers

- Reasons for registering OAuth clients dynamically
- Registering OAuth clients dynamically
- Managing a client registration over time
- Security considerations concerning dynamic OAuth clients
- Protecting dynamic registration with software statements

In OAuth, the client is identified to the authorization server by a *client identifier* that is, generally speaking, unique to the software application functioning as the OAuth client. This client ID is passed in the front end to the authorization endpoint during the authorization request stage of interactive OAuth flows, such as the authorization code grant type that we implemented in chapters 3 through 5. From this client ID, the authorization server can make decisions about which redirect URIs to allow, which scopes to allow, and what information to display to the end user. The client ID is also presented at the token endpoint, and when combined with a client secret the client ID can authenticate the client throughout the authorization delegation process of OAuth.

This client identifier is particularly distinct from any identifiers or accounts that may be held by the resource owner. This distinction is important in OAuth because, you'll recall, OAuth doesn't encourage impersonation of the resource owner. In fact, the entire OAuth protocol is all about acknowledging that a piece of software is acting on behalf of the resource owner. But how does the client get that identifier,

[redacted]

and how does the server know how to associate metadata such as a valid set of redirect URIs or scopes with that identifier?

12.1 How the server knows about the client

In all of our exercises so far, the client ID has been *statically* configured between the authorization server and the client; that is to say, there was an out-of-band agreement—specifically, the text of this book—that determined ahead of time what the client ID and its associated secret would be. The server determined the client ID, which was then copied by hand into the client software. A major downside to this approach is that every instance of every piece of client software for a given API will need to be tied to every instance of an authorization server that protects that API. This is a reasonable expectation when the client and authorization server have a well-established and relatively unchanging relationship, such as when the authorization server is set up to protect a single, proprietary API. For example, back in our cloud-printing example, the user was given the option to import their photos from a specific, well-known photo-storage service. The client was specifically written to talk to this particular service. In this fairly common setup, a limited number of clients will speak to that API and static registration works well enough.

But what if a client is written to access an API that's available across many different servers? What if our printing service could talk to *any* photo-storage service that spoke a standardized photo-storage API? These photo-storage services will likely have separate authorization servers, and such a client will need a client identifier with each one it talks to. We could try to say that we'll reuse the same client ID regardless of the authorization server, but which authorization server gets to pick the ID? After all, not every authorization server will use the same pattern of choosing IDs, and we want to make sure that the chosen ID doesn't conflict with another piece of software on any authorization server. And what happens when there's a new client that needs to be introduced to the ecosystem? Whatever client ID it was assigned would need to be communicated to all authorization servers along with the associated metadata.

Or what if there were many instances of a piece of client software, and each instance needed to talk to the same authorization server? As we saw in chapter 6, this is the case with native applications, and each client instance will need a client identifier to talk to the authorization server. We could once again say that we'll use the same identifier with each instance, and that can work in some cases. But what should we do about a client secret? We already know from chapter 7 that we can't copy the same secret everywhere, since it wouldn't be much of a secret anymore.[1] We

[1] Google famously got around the OAuth 1.0 requirement of every client needing a client secret by declaring that all native applications using Google's OAuth 1.0 server would use the client ID "anonymous" with the client secret "anonymous." This completely breaks the security model's assumptions. Furthermore, Google added an extension parameter to replace the now-missing client ID, which further broke the protocol.

[redacted]

could approach this by leaving the secret out all together and having our client be a public client, which is accepted and codified in the OAuth standard. However, public clients are open to all kinds of attacks including authorization code and token theft as well as impersonation of a genuine client by malicious software. Sometimes, this is an acceptable trade-off, but many times it isn't and we'd like each instance to have a separate client secret.

In either of these cases, manual registration doesn't scale. To put the problem in perspective, consider this extreme but real example: email. Would it be reasonable for a developer to register each copy of an email client with each potential email service provider before shipping the software? After all, every single domain and host on the internet could have its own separate mail server, not to mention intranet mail services. It's clear that this is not at all reasonable, but this is the assumption made with manual registration in OAuth. What if there were another way? Can we introduce clients to authorization servers without manual intervention?

12.2 Registering clients at runtime

The OAuth Dynamic Client Registration protocol[2] provides a way for clients to introduce themselves to authorization servers, including all kinds of information about the client. The authorization server can then provision a unique client ID to the client software that the client can use for all subsequent OAuth transactions, and if appropriate associate a client secret with that ID as well. This protocol can be used by the client software itself, or it could be used as part of a build and deployment system that acts on behalf of the client developer (figure 12.1).

12.2.1 How the protocol works

The core dynamic client registration protocol is a simple HTTP request to the authorization server's client registration endpoint and its corresponding response. This endpoint listens for HTTP POST requests with a JSON body containing a client's proposed metadata information. This call can be optionally protected by an OAuth token, but our example shows an open registration with no authorization.

```
POST /register HTTP/1.1
Host: localhost:9001
Content-Type: application/json
Accept: application/json

{
  "client_name": "OAuth Client",
  "redirect_uris": ["http://localhost:9000/callback"],
  "client_uri": "http://localhost:9000/",
  "grant_types": ["authorization_code"],
  "scope": "foo bar baz"
}
```

[2] RFC 7591 https://tools.ietf.org/html/rfc7591

[redacted]

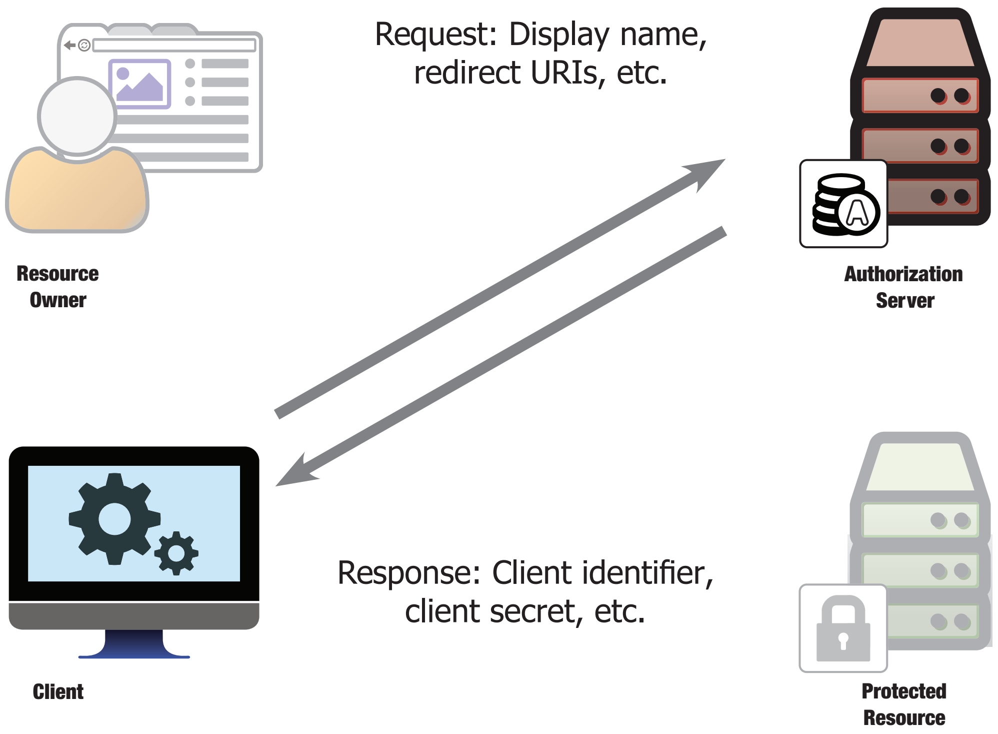

Request: Display name, redirect URIs, etc.
Resource Owner
Authorization Server
Client
Response: Client identifier, client secret, etc.
Protected Resource

Figure 12.1 Information passed in dynamic registration

This metadata can include display names, redirect URIs, scopes, and many other aspects of the client's functionality. (The full official list is included in section 12.3.1, if you'd like to read ahead.) However, the requested metadata can never include a client ID or client secret. Instead, these values are always under the control of the authorization server in order to prevent impersonation of or conflict with other clients IDs, or selection of weak client secrets. The authorization server can perform some basic consistency checks against the presented data, for example, making sure that the requested `grant_types` and `response_types` can be served together, or that the scopes requested are valid for a dynamically registered client. As is generally the case in OAuth, the authorization server gets to make the decisions about what is valid and the client, being a simpler piece of software, obeys what the authorization server dictates.

Upon a successful registration request, the authorization server generates a new client ID and, generally, a client secret. These are sent back to the client along with a copy of the metadata associated with the client. Any values that the client sends in the request are suggested input to the authorization server, but the authorization server has the final say over which values are associated with the client's registration and is free to override or reject any inputs as it sees fit. The resulting registration is sent back to the client as a JSON object.

[redacted]

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "client_id": "1234-wejeg-0392",
  "client_secret": "6trfvbnklp0987trew2345tgvcxcvbjkiou87y6t5r",
  "client_id_issued_at": 2893256800,
  "client_secret_expires_at": 0,
  "token_endpoint_auth_method": "client_secret_basic",
  "client_name": "OAuth Client",
  "redirect_uris": ["http://localhost:9000/callback"],
  "client_uri": "http://localhost:9000/",
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "scope": "foo bar baz"
}
```

In this example, the authorization server has assigned a client ID of `1234-wejeg-0392` and a client secret of `6trfvbnklp0987trew2345tgvcxcvbjkiou87y6t5r` to this client. The client can now store these values and use them for all subsequent communications with the authorization server. Additionally, the authorization server has added a few things to the client's registration record. First, the `token_endpoint_auth_method` value indicates that the client should use HTTP Basic authentication when talking to the token endpoint. Next, the server has filled in the missing `response_types` value to correspond to the `grant_types` value from the client's request. Finally, the server has indicated to the client when the client ID was generated and that the client secret won't expire.

12.2.2 Why use dynamic registration?

There are several compelling reasons for using dynamic registration with OAuth. The original OAuth use cases revolved around single-location APIs, such as those from companies providing web services. These APIs require specialized clients to talk to them, and those clients will need to talk to only a single API provider. In these cases, it doesn't seem unreasonable to expect client developers to put in the effort to register their client with the API, because there's only one provider.

But you've already seen two major exceptions to this pattern wherein these assumptions don't hold. What if there's more than one provider of a given API, or new instances of that same API can be stood up at will? For example, OpenID Connect provides a standardized identity API, and the System for Cross-domain Identity Management (SCIM) protocol provides a standardized provisioning API. Both of these are protected by OAuth, and both can be stood up by different providers. Although a piece of client software could talk to these standard APIs no matter what domains they were running on, we know that managing client IDs in this space is unfeasible. Simply put, writing a new client or deploying a new server for this protocol ecosystem would be a logistical nightmare.

Even if we have a single authorization server to deal with, what about multiple instances of a given client? This is particularly pernicious with native clients on mobile

[redacted]

platforms, because every copy of the client software would have the same client ID and client secret. With dynamic registration, each instance of a client can register itself with the authorization server. Each instance then gets its own client ID and, importantly, its own client secret that it can use to protect its user.

Remember that we said that the kind of interactions email clients have with servers is a driving use case for dynamic registration. Today, OAuth can be used to access Internet Message Access Protocol (IMAP) email services using a Simple Authentication and Security Layer–Generic Security Service Application Program Interface (SASL-GSSAPI) extension. Without dynamic registration, every single mail client would have to preregister with every single possible email provider that allowed OAuth access. This registration would need to be completed by the developer before the software ships, because the end user won't be able to modify and configure the email client once it's installed. The possible combinations are staggering for both authorization servers that need to know about every mail client and for mail clients that need to know about every server. Better, instead, to use dynamic client registration, in which each instance of a mail client can register itself with each instance of an authorization server that it needs to talk to.

Whitelists, blacklists, and graylists

It may seem intimidating to allow dynamic registration on an authorization server. After all, do you want any piece of software to waltz up and start asking for tokens? The truth is, oftentimes you want to do exactly that. Interoperability is by its very nature indistinguishable from an unsolicited request.

Importantly, a client being registered at the authorization server doesn't entitle that client to access any resources protected by that authorization server. Instead, a resource owner still needs to delegate some form of access to the client itself. This key fact differentiates OAuth from other security protocols wherein the registration event carries with it authority to access resources and therefore needs to be protected by a strict onboarding process.

For clients that have been vetted by administrators of the authorization server, and have been statically registered by such trusted authorities, the authorization server might want to skip prompting the resource owner for their consent. By placing certain trusted clients on a *whitelist*, the authorization server can smooth the user experience for these clients. The OAuth protocol works exactly the same as before: the resource owner is redirected to the authorization endpoint, where they authenticate, and the authorization server reads the access request in the front channel. But instead of prompting the user for their decision about a trusted client, the authorization server can have policy decide that a client is already authorized, and return the result of the authorization request immediately.

At the other end of the spectrum, an authorization server can decide that it never wants to let clients with particular attributes register or request authorization. This

(continued)

[redacted]

can be a set of redirect URIs that are known to house malicious software, or display names that are known to be intentionally confusing to end users, or other types of detectable malice. By placing these attribute values on a *blacklist*, the authorization server can prevent clients from ever using them.

Everything else can go on a *graylist*, whereby resource owners make the final authorization decisions. Dynamically registered clients that don't fall under the blacklist and have not yet been whitelisted should automatically fall on the graylist. These clients can be more limited than statically registered clients, such as not having the ability to ask for certain scopes or use certain grant types, but otherwise they function as normal OAuth clients. This will allow greater scalability and flexibility for an authorization server without compromising its security posture. A dynamically registered client used successfully by many users over a sufficiently long period of time could eventually become whitelisted, and malicious clients could find their registrations revoked and their key attributes blacklisted.

12.2.3 Implementing the registration endpoint

Now that you know how the protocol works, we're going to implement it. We'll start on the server side with the registration endpoint. Open up `ch-12-ex-1` and edit `authorizationServer.js` for this part of the exercise. Our authorization server will use the same in-memory array for client functionality that it did in chapter 5, meaning this storage will reset whenever the server is restarted. In contrast, a production environment is likely to want to use a database or some other, more robust storage mechanism.

First, we'll need to create the registration endpoint. On our server, this listens for HTTP POST requests on the `/register` URL, so we'll set up a handler for that. In our server, we're only going to be implementing public registration, which means we're not going to be requiring the optional OAuth access token at our registration endpoint. We're also going to set up a variable to collect the incoming client metadata requests as we process them.

```
app.post('/register', function (req, res){
   var reg = {};
});
```

The Express.js code framework in our application is set up to automatically parse the incoming message as a JSON object, which is made available to the code in the `req.body` variable. We're going to do a handful of basic consistency checks on the incoming data. First, we'll see what the client has asked for as an authentication method. If it hasn't specified one, we're going to default to using a client secret over HTTP Basic. Otherwise, we'll take the input value specified by the client. We'll then check to make sure that value is valid, and return an `invalid_client_metadata` error if it is not. Note that the values for this field, like `secret_basic`, are defined by the specification and can be extended with new definitions.

[redacted]

```
if (!req.body.token_endpoint_auth_method) {
   reg.token_endpoint_auth_method = 'secret_basic';
} else {
   reg.token_endpoint_auth_method = req.body.token_endpoint_auth_method;
}

if (!__.contains(['secret_basic', 'secret_post', 'none'],
reg.token_endpoint_auth_method)) {
   res.status(400).json({error: 'invalid_client_metadata'});
   return;
}
```

Next, we'll read in the `grant_type` and `response_type` values and ensure that they are consistent. If the client doesn't specify either, we'll default them to an authorization code grant. If they request a `grant_type` but not its corresponding `response_type`, or vice versa, we'll fill in the missing value for them. The specification defines not only the appropriate values but also the relationship between these two values. Our simple server supports only the authorization code and refresh token grants, so we're going to send back an `invalid_client_metadata` error if they request anything else.

```
if (!req.body.grant_types) {
   if (!req.body.response_types) {
        reg.grant_types = ['authorization_code'];
        reg.response_types = ['code'];
   } else {
        reg.response_types = req.body.response_types;
        if (__.contains(req.body.response_types, 'code')) {
               reg.grant_types = ['authorization_code'];
        } else {
               reg.grant_types = [];
        }
   }
} else {
   if (!req.body.response_types) {
        reg.grant_types = req.body.grant_types;
        if (__.contains(req.body.grant_types, 'authorization_code')) {
               reg.response_types =['code'];
        } else {
               reg.response_types = [];
        }
   } else {
        reg.grant_types = req.body.grant_types;
        reg.reponse_types = req.body.response_types;
        if (__.contains(req.body.grant_types, 'authorization_code') && !__.
   contains(req.body.response_types, 'code')) {
               reg.response_types.push('code');
        }
        if (!__.contains(req.body.grant_types, 'authorization_code') &&
   __.contains(req.body.response_types, 'code')) {
               reg.grant_types.push('authorization_code');
        }
   }
}
```

[redacted]

```
if (!__.isEmpty(__.without(reg.grant_types, 'authorization_code',
'refresh_token')) ||
       !__.isEmpty(__.without(reg.response_types, 'code'))) {
  res.status(400).json({error: 'invalid_client_metadata'});
  return;
}
```

Next, we'll make sure that the client has registered at least one redirect URI. We enforce this on all clients because this version of the server supports only the authorization code grant type, which is based on a redirect. If you were supporting other grant types that don't use redirection, you'd want to make this check conditional on the grant type. If you were checking redirect URIs against a blacklist, this would be a good place to implement that functionality as well, but implementing that type of filtering is left as an exercise to the reader.

```
if (!req.body.redirect_uris || !__.isArray(req.body.redirect_uris) ||
__.isEmpty(req.body.redirect_uris)) {
  res.status(400).json({error: 'invalid_redirect_uri'});
  return;
} else {
  reg.redirect_uris = req.body.redirect_uris;
}
```

Next, we'll copy over the other fields that we care about, checking their data types on the way. Our implementation will ignore any additional fields passed in that it doesn't understand, though a production quality implementation might want to hang on to extra fields in case additional functionality is added to the server at a later time.

```
if (typeof(req.body.client_name) == 'string') {
  reg.client_name = req.body.client_name;
}

if (typeof(req.body.client_uri) == 'string') {
  reg.client_uri = req.body.client_uri;
}

if (typeof(req.body.logo_uri) == 'string') {
  reg.logo_uri = req.body.logo_uri;
}

if (typeof(req.body.scope) == 'string') {
  reg.scope = req.body.scope;
}
```

Finally, we'll generate a client ID and, if the client is using an appropriate token endpoint authentication method, a client secret. We'll also note the registration timestamp and mark that the secret doesn't expire. We'll attach these directly to our client registration object that we've been building up.

```
reg.client_id = randomstring.generate();
if (__.contains(['client_secret_basic', 'client_secret_post']),
  reg.token_endpoint_auth_method) {
  reg.client_secret = randomstring.generate();
}
```

[redacted]

```
reg.client_id_created_at = Math.floor(Date.now() / 1000);
reg.client_secret_expires_at = 0;
```

Now we can store that client object in our client storage. As a reminder, we're using a simple in-memory array, but a production system would probably be using a database for this part. After we've stored it, we return the JSON object to the client.

```
clients.push(reg);

res.status(201).json(reg);
return;
```

Once it's all put together, our registration endpoint looks like listing 13 in appendix B.

Our authorization server's registration system is simple, but it can be augmented to do other checks on the client such as checking all URLs against a blacklist, limiting the scopes available to dynamically registered clients, ensuring the client provides a contact address, or any number of other checks. The registration endpoint can also be protected by an OAuth token, thereby associating the registration with the resource owner who authorized that token. These enhancements are left as an exercise to the reader.

12.2.4 Having a client register itself

Now we're going to set up our client to register itself as needed. Using the previous exercise, edit `client.js`. Near the top of the file, note that we've set aside an empty object for storing client information:

```
var client = {};
```

Instead of filling it in by hand as we did in chapter 3, we're going to use the dynamic registration protocol. Once again, this is an in-memory storage solution that gets reset every time the client software is restarted, and a production system will likely use a database or other storage mechanism for this role.

First, we have to decide whether we need to register, because we don't want to register a new client every time we need to talk to the authorization server. When the client is about to send the initial authorization request, it first checks to see whether it has a client ID for the authorization server. If it doesn't have one, it calls a utility function to handle client registration. If the registration was successful, the client continues on. If it wasn't, the client renders an error and gives up. This code is already included in the client.

```
if (!client.client_id) {
   registerClient();
   if (!client.client_id) {
        res.render('error', {error: 'Unable to register client.'});
        return;
   }
}
```

Now we'll be defining that `registerClient` utility function. This is a simple function that will `POST` a registration request to the authorization server and store the response in the `client` object.

[redacted]

```
var registerClient = function() {

};
```

First, we need to define the metadata values that we're sending to the authorization server. These act as a kind of template for our client's configuration, and the authorization server will fill in the other required fields such as the client ID and client secret for us.

```
var template = {
  client_name: 'OAuth in Action Dynamic Test Client',
  client_uri: 'http://localhost:9000/',
  redirect_uris: ['http://localhost:9000/callback'],
  grant_types: ['authorization_code'],
  response_types: ['code'],
  token_endpoint_auth_method: 'secret_basic'
};
```

We send this template object over to the server in an HTTP POST request.

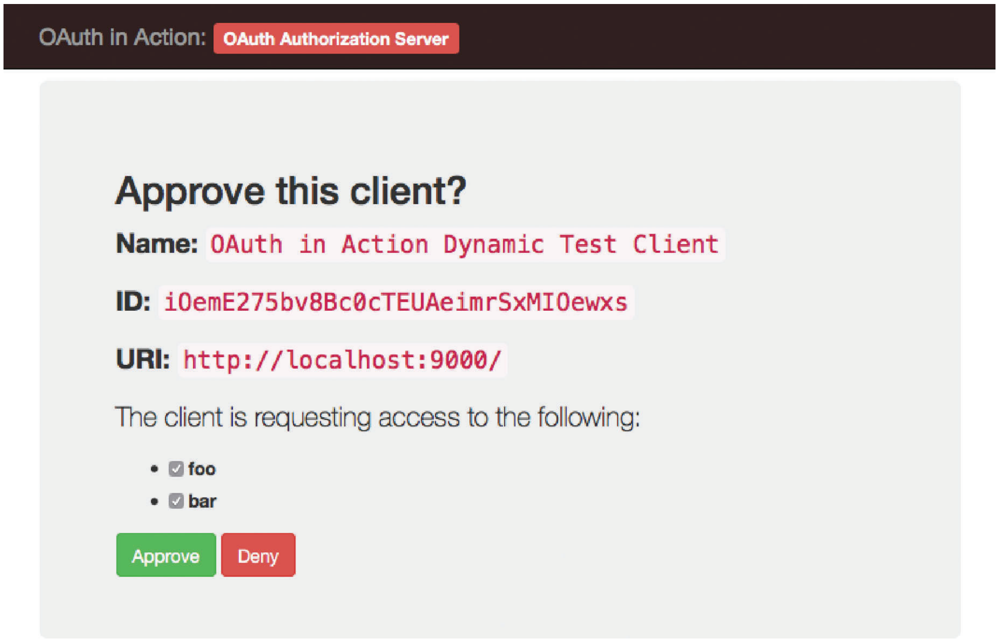
```
var headers = {
  'Content-Type': 'application/json',
  'Accept': 'application/json'
};

var regRes = request('POST', authServer.registrationEndpoint,
  {
       body: JSON.stringify(template),
       headers: headers
  }
);
```

Now we'll check the result object. If we get back an HTTP code of 201 Created, we save the returned object to our client object. If we get back an error of any kind, we don't save the client object, and we allow whatever function called us to pick up on the error state of the client being unregistered and handle it appropriately.

```
if (regRes.statusCode == 201) {
  var body = JSON.parse(regRes.getBody());
  console.log("Got registered client", body);
  if (body.client_id) {
       client = body;
  }
}
```

From here, the rest of the application takes over as normal. No further changes are needed to the calls to the authorization server, the processing of tokens, or access of the protected resource (figure 12.2). The client name that you registered now shows up on the authorization screen, as does the dynamically generated client ID. To test this, edit the client's `template` object, restart the client, and run the test again. Note that you don't need to restart the authorization server for the registration to succeed. Since the authorization server can't identify the client software making the request, it will happily accept a new registration request from the same client software many times, issuing a new client ID and secret each time.

[redacted]


OAuth in Action: OAuth Authorization Server

Approve this client?

Name: OAuth in Action Dynamic Test Client

ID: i0emE275bv8Bc0cTEUAeimrSxMI0ewxs

URI: http://localhost:9000/

The client is requesting access to the following:

- ✓ foo
- ✓ bar

Figure 12.2 The authorization server's approval page, showing a randomized client ID and the requested client display name

Some clients need to be able to get tokens from more than one authorization server. For an additional exercise, refactor the client's storage of its registration information such that it's dependent on the authorization server that the client is talking to. For an additional challenge, implement this using a persistent database instead of an in-memory storage mechanism.

12.3 Client metadata

The attributes associated with a registered client are collectively known as its *client metadata*. These attributes include those that affect the functionality of the underlying protocol, such as `redirect_uris` and `token_endpoint_auth_method`, as well as those that affect the user experience, such as `client_name` and `logo_uri`. As you've seen in the previous examples, these attributes are used in two different ways in the dynamic registration protocol:

1. *Client sending to server.* The client sends a set of *requested* attribute values to the authorization server. These requested values may not be compatible with the configuration of a given authorization server, such as a request for `grant_types` that the server doesn't support or `scope` that the client isn't authorized for, so the client must not always expect a successful registration to match what it has requested.

[redacted]

2. *Server returning to client.* The authorization server sends back a set of *registered* attribute values to the client. The authorization server can replace, augment, or remove the client's requested attributes however it sees fit. The authorization server will usually attempt to honor the client's requested attributes, but the authorization server always has the final say. In any event, the authorization server must always return the actual registered attributes to the client. The client can respond to an undesirable registration outcome however it sees fit, including attempting to modify the registration with more applicable values or refusing to communicate with that authorization server.

As in most places in OAuth, the client is subservient to the authorization server. The client can request, but the authorization server dictates the final reality.

12.3.1 Table of core client metadata field names

The core dynamic client registration protocol defines a set of common client metadata names, and this set can be extended. For example, the OpenID Connect Dynamic Client Registration specification, which is based on and compatible with OAuth Dynamic Client Registration, extends this list with a few more of its own, specific to the OpenID Connect protocol that we'll cover chapter 13. We've included a few OpenID Connect specific extensions in table 12.1 that have general applicability to OAuth clients.

12.3.2 Internationalization of human-readable client metadata

Among the various possible pieces of client information that are sent in a registration request and response, several are intended to be presented to the resource owner on the authorization page or other user-facing screens on the authorization server. These consist of either strings that are displayed directly for the user (such as `client_name`, the display name of the client software) or URLs for the user to click on (such as `client_uri`, the client's homepage). But if a client can be used in more than one language or locale, it could have a version of these human-readable values for each language that it supports. Would such a client need to register separately for each language family?

Thankfully not, as the dynamic client registration protocol has a system (borrowed from OpenID Connect) for representing values in multiple languages simultaneously. In a plain claim, such as `client_name`, the field and value will be stored as a normal JSON object member:

```
"client_name": "My Client"
```

In order to represent a different language or script, the client also sends a version of the field with the language tag appended to the field name with the # (pound or hash) character. For example, let's say that this client is known as "Mon Client" in French. The language code for French is `fr`, and so the field would be represented as `client_name#fr` in the JSON object. These two fields would be sent together.

```
"client_name": "My Client",
"client_name#fr": "Mon Client"
```

[redacted]

Table 12.1 Client metadata fields available in dynamic client registration

Name	Values and Description	
redirect_uris	An array of URI strings used in redirect-based OAuth grants, such as authorization_code and implicit.	
token_endpoint_auth_method	How the client will authenticate to the token endpoint.	
	none	The client doesn't authenticate to the token endpoint, either because it doesn't use the token endpoint, or it uses the token endpoint but is a public client.
	client_secret_basic	The client sends its client secret using HTTP Basic. This is the default value if one isn't specified and the client is issued a client secret.
	client_secret_post	The client sends its client secret using HTTP form parameters.
	client_secret_jwt	The client will create a JSON Web Token (JWT) symmetrically signed with its client secret.
	private_key_jwt	The client will create a JWT asymmetrically signed with its private key. The public key will need to be registered with the authorization server.
grant_types	Which grant types the client will use to get tokens. The values here are the same ones used at the token endpoint in the grant_type parameter.	
	authorization_code	The authorization code grant, where the client sends the resource owner to the authorization endpoint to obtain an authorization code, then presents that code back to the token endpoint. Needs to be used with the "code" response_type.
	implicit	The implicit grant, where the client sends the resource owner to the authorization endpoint to obtain a token directly. Needs to be used with the "token" response type.
	password	The resource owner password grant, where the client prompts the resource owner for their username and password and exchanges them for a token at the token endpoint.
	client_credentials	The client credentials grant, where the client uses its own credentials to obtain a token for itself.
	refresh_token	The refresh token grant, where the client uses a refresh token to obtain a new access token when the resource owner is no longer present.
	urn:ietf:params:oauth:grant-type:jwt-bearer	The JWT assertion grant, where the client presents a JWT with specific claims to obtain a token.

[redacted]

Table 12.1 Client metadata fields available in dynamic client registration (*continued*)

Name	Values and Description	
	`urn:ietf:params:oauth:grant-type:saml2-bearer`	The Security Assertion Markup Language (SAML) assertion grant, where the client presents a SAML document with specific claims to obtain a token.
`response_types`	Which response types the client will use at the authorization endpoint. These values are the same as those used in the `response_type` parameter.	
	`code`	The authorization code response type, which returns an authorization code that needs to be handed in at the token endpoint to get a token.
	`token`	The implicit response type, which returns a token directly to the redirect URI.
`client_name`	A human-readable display name for the client.	
`client_uri`	A URI that indicates the client's homepage.	
`logo_uri`	A URI for a graphical logo for the client. The authorization server can use this URL to display a logo for the client to the user, but keep in mind that fetching an image URL could have security and privacy considerations for the user.	
`scope`	A list of scopes that the client can use when requesting tokens. This is formatted as a string of space-separated values, as in the OAuth protocol.	
`contacts`	A list of ways to contact the people responsible for a client. Usually these are email addresses, but they could be phone numbers, instant messaging addresses, or other contact mechanisms.	
`tos_uri`	A URI for a human-readable page that lists the terms of service for the client. These terms describe the contractual relationship that the resource owner accepts when authorizing the client.	
`policy_uri`	A URI for a human-readable page that contains the privacy policy for the client. This policy describes how the organization that has deployed the client collects, uses, retains, and discloses the resource owner's personal data, including data accessed through authorized API calls.	
`jwks_uri`	A URI that points to the JSON Web Key Set containing the public keys for this client, hosted in a place accessible to the authorization server. This field can't be used along with the `jwks` field. The `jwks_uri` field is preferred, as it allows the client to rotate keys.	
`jwks`	A JSON Web Key Set document (a JSON object) containing the public keys for this client. This field can't be used along with the `jwks_uri` field. The `jwks_uri` field is preferred, as it allows the client to rotate keys.	
`software_id`	A unique identifier for the software that the client is running. This identifier will be the same across all instances of a given piece of client software.	
`software_version`	A version identifier for the client software indicated by the `software_id` field. The version string is opaque to the authorization server, and no particular format is assumed.	

[redacted]

The authorization server should use the most specific entry possible in interacting with users. For instance, if a user on the authorization server has registered their preferred language as French, the authorization server would display the French version of the name in lieu of the generic version. The client should always provide the generic version of a field name, because if nothing more specific is available, or if international locales aren't supported, the authorization server will display the generic text with no locale qualifier.

Implementation and use of this feature are left as an exercise to the reader, since it requires a bit of fiddling with the client's data model and the web server's locale settings to be useful. Although some programming languages are able to automatically parse JSON objects into native objects in the language platform, and thereby offer native object-member access to the values, the `#` character used in this internationalization method is often not a valid character for object method names. Therefore, alternative access methods need to be used. For example, in JavaScript, the first value in the previous object could be accessed as `client.client_name`, but the second value would need to be accessed as `client[“client_name#fr”]` instead.

12.3.3 Software statements

Every metadata value that a client sends in a dynamic registration request needs to be considered completely self-asserted. In such circumstances, there is nothing preventing a client from claiming a misleading client name or a redirect URI on someone else's domain. As you saw in chapters 7 and 9, this can lead to a whole range of vulnerabilities if the authorization server isn't careful.

But what if we had a way to present client metadata to the authorization server in a way that the authorization server could verify that it's coming from a trusted party? With such a mechanism, the authorization server would be able to lock down certain metadata attributes in clients and have a higher assurance that the metadata is valid. The OAuth dynamic registration protocol provides such a mechanism in the *software statement*.

Simply put, a software statement is a signed JWT that contains as its payload client metadata as it would be found inside a request to the registration endpoint, as we saw in section 12.2. Instead of manually registering each instance of a piece of client software with all authorization servers, the client developer can instead preregister some subset of their client's metadata, particularly the subset not likely to change over time, at a trusted third party and be issued a software statement signed by the trusted party. The client software can then present this software statement along with any additional metadata required for registration to the authorization servers that it registers at.

Let's take a look at a concrete example. Suppose that a developer wants to preregister a client such that the client name, client homepage, logo, and terms of service are constant across all instances of the client and across all authorization servers. The developer registers these fields with a trusted authority and is issued a software statement in the form of a signed JWT.

[redacted]

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzb2Z0d2FyZV9pZCI6Ijg0MDEyLTM5MTM0LTM5
MTIiLCJzb2Z0d2FyZV92ZXJzaW9uIjoiMS4yLjUtZG9scGhpbiIsImNsaWVudF9uYW1lIjoiU3BlY
2lhbCBPQXV0aCBDbGllbnQiLCJjbGllbnRfdXJpIjoiaHR0cHM6Ly9leGFtcGxlLm9yZy8iLCJsb2
dvX3VyaSI6Imh0dHBzOi8vZXhhbXBsZS5vcmcvbG9nby5wbmciLCJ0b3NfdXJpIjoiaHR0cHM6Ly9
leGFtcGxlLm9yZy90ZXJtcy1vZi1zZXJ2aWNlLyJ9.X4k7X-JLnOM9rZdVugYgHJBBnq3s9RsugxZ
QHMfrjCo
```

The payload of this JWT decodes into a JSON object much like one that would be sent in a registration request.

```
{
  "software_id": "84012-39134-3912",
  "software_version": "1.2.5-dolphin",
  "client_name": "Special OAuth Client",
  "client_uri": "https://example.org/",
  "logo_uri": "https://example.org/logo.png",
  "tos_uri": "https://example.org/terms-of-service/"
}
```

The registration request sent by the client can contain additional fields not found in the software statement. In this example, client software can be installed on different hosts, necessitating different redirect URIs, and be configured to access different scopes. A registration request for this client would include its software statement as an additional parameter.

```
POST /register HTTP/1.1
Host: localhost:9001
Content-Type: application/json
Accept: application/json

{
  "redirect_uris": ["http://localhost:9000/callback"],
  "scope": "foo bar baz",
  "software_statement": " eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzb2Z0d2FyZV
   9pZCI6Ijg0MDEyLTM5MTM0LTM5MTIiLCJzb2Z0d2FyZV92ZXJzaW9uIjoiMS4yLjUtZG9scGhp
   biIsImNsaWVudF9uYW1lIjoiU3BlY2lhbCBPQXV0aCBDbGllbnQiLCJjbGllbnRfdXJpIjoiaH
   R0cHM6Ly9leGFtcGxlLm9yZy8iLCJsb2dvX3VyaSI6Imh0dHBzOi8vZXhhbXBsZS5vcmcvbG9n
   by5wbmciLCJ0b3NfdXJpIjoiaHR0cHM6Ly9leGFtcGxlLm9yZy90ZXJtcy1vZi1zZXJ2aWNlLy
   J9.X4k7X-JLnOM9rZdVugYgHJBBnq3s9RsugxZQHMfrjCo"
}
```

The authorization server will parse the software statement, validate its signature, and determine that it has been issued by an authority it trusts. If so, the claims inside the software statement will supersede those presented in the unsigned JSON object.

Software statements allow for a level of trust beyond the self-asserted values usually found in OAuth. They also allow a network of authorization servers to trust a central authority (or several) to issue software statements for different clients. Furthermore, multiple instances of a client can be logically grouped together at the authorization server by the information in the software statement that they all present. Although each instance would still get its own client ID and client secret, a server administrator

[redacted]

could have an option to disable or revoke all copies of a given piece of software at once in the case of malicious behavior on the part of any instance.

Implementation of software statements is left as an exercise to the reader.

12.4 Managing dynamically registered clients

A client's metadata doesn't always remain static over time. Clients could change their display names, add or remove redirect URIs, require a new scope for new functionality, or make any number of other changes over the lifetime of the client. The client might want to read its own configuration, too. If the authorization server rotates a client's secret after some amount of time or a triggering event, the client will need to know the new secret. Finally, if a client knows that it's not going to be used again, such as when it's getting uninstalled by the user, it can tell the authorization server to get rid of its client ID and associated data.

12.4.1 How the management protocol works

For all of these use cases, the OAuth Dynamic Client Registration Management protocol[3] defines a RESTful protocol extension to OAuth Dynamic Client Registration. The management protocol extends the core registration protocol's "create" method with associated "read," "update," and "delete" methods, allowing for full lifecycle management of dynamically registered clients.

To accomplish this, the management protocol extends the response from the registration endpoint with two additional fields. First, the server sends the client a client configuration endpoint URI in the `registration_client_uri` field. This URI provides all of the management functionality for this specific client. The client uses this URI directly as it's given, with no additional parameters or transformations required. It's often unique for each client registered at an authorization server, but it's entirely up to the authorization server to decide how the URI itself is structured. Second, the server also sends a specialized access token, called a registration access token, in the `registration_access_token` field. This is an OAuth bearer token that the client can use to access the client configuration endpoint, and nowhere else. As with all other OAuth tokens, it's entirely up to the authorization server what the format of this token is, and the client uses it as given.

Let's take a look at a concrete example. First, the client sends the same registration request to the registration endpoint that was made in the example in section 12.1.3. The server responds in the same way, except that the JSON object is extended as we discussed. Our authorization server creates the client configuration endpoint URI by concatenating the client ID to the registration endpoint, in keeping with common RESTful design principles, but the authorization server is free to format this URL however it pleases. The registration access token in our server is another randomized string like other tokens we generate.

[3] RFC 7592 https://tools.ietf.org/html/rfc7592

[redacted]

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "client_id": "1234-wejeg-0392",
  "client_secret": "6trfvbnklp0987trew2345tgvcxcvbjkiou87y6t5r",
  "client_id_issued_at": 2893256800,
  "client_secret_expires_at": 0,
  "token_endpoint_auth_method": "client_secret_basic",
  "client_name": "OAuth Client",
  "redirect_uris": ["http://localhost:9000/callback"],
  "client_uri": "http://localhost:9000/",
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "scope": "foo bar baz",
  "registration_client_uri": "http://localhost:9001/register/1234-wejeg-0392"
  "registration_access_token": "ogh238fj2f0zFaj38dA"
}
```

The remainder of the registration response is the same as it was earlier. If the client wants to read its registration information, it sends an HTTP GET request to the client configuration endpoint, using the registration access token in the authorization header.

```
GET /register/1234-wejeg-0392 HTTP/1.1
Accept: application/json
Authorization: Bearer ogh238fj2f0zFaj38dA
```

The authorization server checks to make sure that the client referred to in the configuration endpoint URI is the same one that the registration access token was issued to. As long as everything is valid, the server responds similarly to a normal registration request. The body is still a JSON object describing the registered client, but the response code is now an HTTP 200 OK message instead. The authorization server is free to update any of the client's fields, including the client secret and registration access token, but the client ID doesn't change. In this example, the server has rotated the client's secret for a new one, but all other values remain the same. Note that the response includes the client configuration endpoint URI as well as the registration access token.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "client_id": "1234-wejeg-0392",
  "client_secret": "6trfvbnklp0987trew2345tgvcxcvbjkiou87y6",
  "client_id_issued_at": 2893256800,
  "client_secret_expires_at": 0,
  "token_endpoint_auth_method": "client_secret_basic",
  "client_name": "OAuth Client",
  "redirect_uris": ["http://localhost:9000/callback"],
  "client_uri": "http://localhost:9000/",
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
```

[redacted]

```
  "scope": "foo bar baz",
  "registration_client_uri": "http://localhost:9001/register/1234-wejeg-0392"
  "registration_access_token": "ogh238fj2f0zFaj38dA"
}
```

If the client wants to be able to update its registration, it sends an HTTP PUT request to the configuration endpoint, again using the registration access token in the authorization header. The client includes its entire configuration as returned from its registration request, including its previously issued client ID and client secret. However, just as in the initial dynamic registration request, the client cannot choose its own values for its client ID or client secret. The client also doesn't include the following fields (or their associated values) in its request:

- `client_id_issued_at`
- `client_secret_expires_at`
- `registration_client_uri`
- `registration_access_token`

All other values in the request object are requests to replace existing values in the client's registration. Fields that are left out of the request are interpreted to be deletions of existing values.

```
PUT /register/1234-wejeg-0392 HTTP/1.1
Host: localhost:9001
Content-Type: application/json
Accept: application/json
Authorization: Bearer ogh238fj2f0zFaj38dA

{
  "client_id": "1234-wejeg-0392",
  "client_secret": "6trfvbnklp0987trew2345tgvcxcvbjkiou87y6",
  "client_name": "OAuth Client, Revisited",
  "redirect_uris": ["http://localhost:9000/callback"],
  "client_uri": "http://localhost:9000/",
  "grant_types": ["authorization_code"],
  "scope": "foo bar baz"
}
```

The authorization server once again checks to make sure that the client referred to in the configuration endpoint URI is the same one that the registration access token was issued to. The authorization server will also check the client secret, if included, to make sure that it matches the expected value. The authorization server responds with a message identical to the one from a read request, an HTTP 200 OK with the details of the registered client in a JSON object in the body. The authorization server is free to reject or replace any input from the client as it sees fit, as with the initial registration request. The authorization server is once again able to change any of the client's metadata information, except for its client ID.

If the client wants to unregister itself from the authorization server, it sends an HTTP DELETE request to the client configuration endpoint with the registration access token in the authorization header.

[redacted]

```
DELETE /register/1234-wejeg-0392 HTTP/1.1
Host: localhost:9001
Authorization: Bearer ogh238fj2f0zFaj38dA
```

The authorization server yet again checks to make sure that the client referred to in the configuration endpoint URI is the same one that the registration access token was issued to. If they match, and if the server is able to delete the client, it responds with an empty HTTP 204 No Content message.

```
HTTP/1.1 204 No Content
```

From there, the client needs to discard its registration information including its client ID, client secret, and registration access token. The authorization server should also, if possible, delete all access and refresh tokens associated with the now-deleted client.

12.4.2 Implementing the dynamic client registration management API

Now that we know what's expected for each action, we're going to implement the management API in our authorization server. Open up `ch-12-ex-2` and edit the `authorizationServer.js` file for this exercise. We've provided an implementation of the core dynamic client registration protocol already, so we'll be focusing on the new functionality needed to support the management protocol. Remember, if you want, you can view all registered clients by visiting the authorization server's homepage at `http://localhost:9001/` where it will print out all client information for all registered clients (figure 12.3).

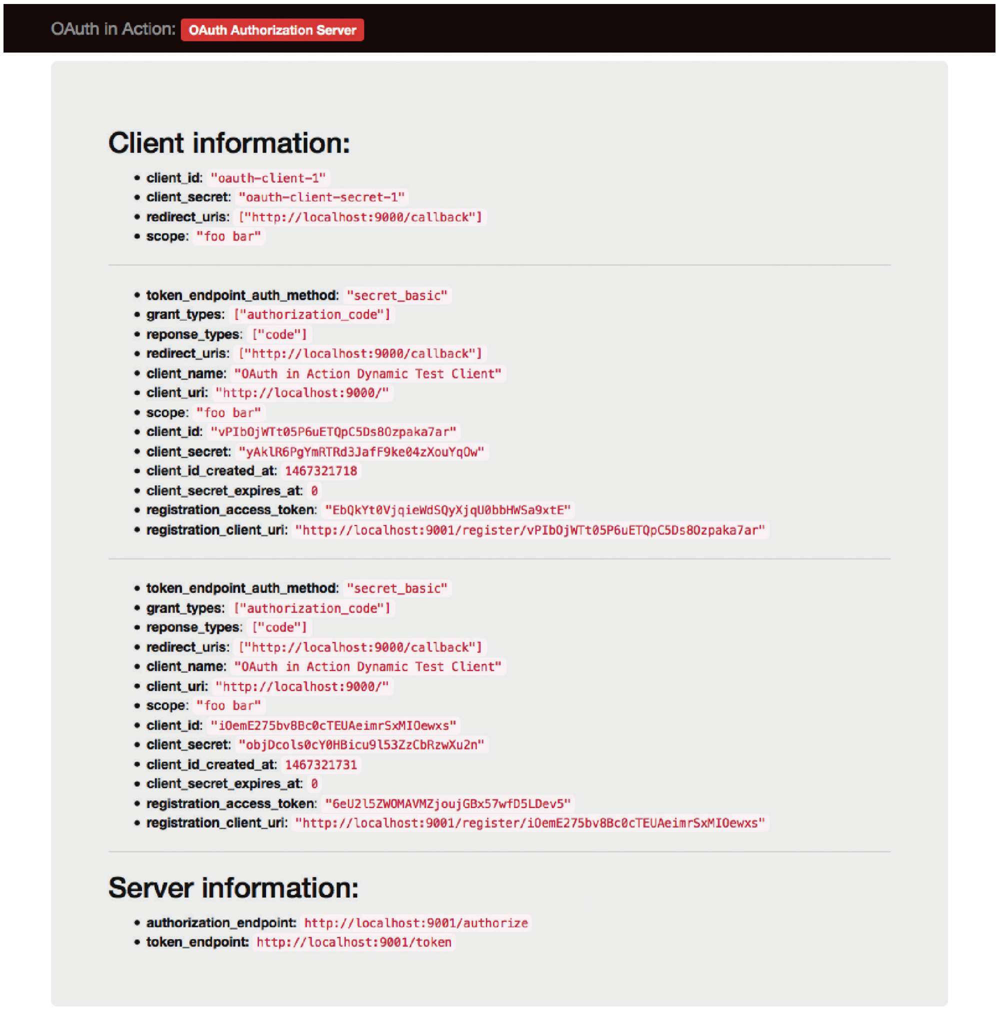
In the registration handler function, the first thing you may notice is that we've abstracted out the client metadata checks from the exercise in section 12.1 into a utility function. This was done so that we could reuse the same checks in several functions. If the requested metadata passes all checks, it's returned. If any of the checks fails, the utility function sends the appropriate error response over the HTTP channel and returns `null`, leaving the calling function to return immediately with no further handling required. In the registration function, when we call this check, we now do this:

```
var reg = checkClientMetadata(req);
if (!reg) {
  return;
}
```

First, we'll need to augment the client information that's returned from the registration endpoint. Right after we generate the client ID and secret, but before we render the output in the response, we need to create a registration access token and attach it to the client object to be checked later. We'll also need to generate and return the client configuration endpoint URI, which in our server will be made by appending the client ID to the registration endpoint's URI.

```
reg.client_id = randomstring.generate();
if (__.contains(['client_secret_basic', 'client_secret_post']),
reg.token_endpoint_auth_method) {
  reg.client_secret = randomstring.generate();
}
```

[redacted]



OAuth in Action: OAuth Authorization Server

Client information:

- **client_id**: `"oauth-client-1"`
- **client_secret**: `"oauth-client-secret-1"`
- **redirect_uris**: `["http://localhost:9000/callback"]`
- **scope**: `"foo bar"`

- **token_endpoint_auth_method**: `"secret_basic"`
- **grant_types**: `["authorization_code"]`
- **reponse_types**: `["code"]`
- **redirect_uris**: `["http://localhost:9000/callback"]`
- **client_name**: `"OAuth in Action Dynamic Test Client"`
- **client_uri**: `"http://localhost:9000/"`
- **scope**: `"foo bar"`
- **client_id**: `"vPIbOjWTt05P6uETQpC5Ds8Ozpaka7ar"`
- **client_secret**: `"yAklR6PgYmRTRd3JafF9ke04zXouYqOw"`
- **client_id_created_at**: `1467321718`
- **client_secret_expires_at**: `0`
- **registration_access_token**: `"EbQkYt0VjqieWdSQyXjqU0bbHWSa9xtE"`
- **registration_client_uri**: `"http://localhost:9001/register/vPIbOjWTt05P6uETQpC5Ds8Ozpaka7ar"`

- **token_endpoint_auth_method**: `"secret_basic"`
- **grant_types**: `["authorization_code"]`
- **reponse_types**: `["code"]`
- **redirect_uris**: `["http://localhost:9000/callback"]`
- **client_name**: `"OAuth in Action Dynamic Test Client"`
- **client_uri**: `"http://localhost:9000/"`
- **scope**: `"foo bar"`
- **client_id**: `"iOemE275bv8Bc0cTEUAeimrSxMIOewxs"`
- **client_secret**: `"objDcols0cY0HBicu9l53ZzCbRzwXu2n"`
- **client_id_created_at**: `1467321731`
- **client_secret_expires_at**: `0`
- **registration_access_token**: `"6eU2l5ZWOMAVMZjoujGBx57wfD5LDev5"`
- **registration_client_uri**: `"http://localhost:9001/register/iOemE275bv8Bc0cTEUAeimrSxMIOewxs"`

Server information:

- **authorization_endpoint**: `http://localhost:9001/authorize`
- **token_endpoint**: `http://localhost:9001/token`

Figure 12.3 The authorization server showing several registered clients

```
reg.client_id_created_at = Math.floor(Date.now() / 1000);
reg.client_secret_expires_at = 0;

reg.registration_access_token = randomString.generate();
reg.registration_client_uri = 'http://localhost:9001/register/' + reg.client_id;

clients.push(reg);

res.status(201).json(reg);
return;
```

Now both the stored client information and the returned JSON object contain the access token and the client registration URI. Next, because we're going to need to check the registration access token against every request to the management API,

[redacted]

we're going to create a filter function to handle that common code. Remember that this filter function takes in a third parameter, `next`, which is the function to call after the filter has run successfully.

```
var authorizeConfigurationEndpointRequest = function(req, res, next) {

};
```

First, we'll take the client ID off the incoming request URL and try to look up the client. If we can't find it, return an error and bail.

```
var clientId = req.params.clientId;
var client = getClient(clientId);
if (!client) {
   res.status(404).end();
   return;
}
```

Next, parse the registration access token from the request. Although we're free to use any valid methods for bearer token presentation here, we're going to limit things to the authorization header for simplicity. As we do in the protected resource, check the authorization header and look for the bearer token. If we don't find it, return an error.

```
var auth = req.headers['authorization'];
if (auth && auth.toLowerCase().indexOf('bearer') == 0) {
   var regToken = auth.slice('bearer '.length);
} else {
   res.status(401).end();
   return;
}
```

we found a registration access token on the request, need to process it below

Finally, if we do get an access token, we need to make sure it's the right token for this registered client. If it matches, we can continue to the next function in the handler chain. Since we've already looked up the client, we attach it to the request so we don't have to look it up again. If the token doesn't match, we return an error.

```
if (regToken == client.registration_access_token) {
   req.client = client;
   next();
   return;
} else {
   res.status(403).end();
   return;
}
```

Now we can start to tackle the functionality. First, we'll set up handlers for all three functions, making sure to add the filter function to the handler setup. Each of these sets up the special `:clientId` path element, which is parsed by the Express.js framework and handed to us in the `req.params.clientId` variable, as used in the previous filter function.

```
app.get('/register/:clientId', authorizeConfigurationEndpointRequest,
function(req, res) {

});
```

[redacted]

```
app.put('/register/:clientId', authorizeConfigurationEndpointRequest,
function(req, res) {

});

app.delete('/register/:clientId', authorizeConfigurationEndpointRequest,
function(req, res) {

});
```

Let's start with the read function first. Since the filter function has already validated the registration access token and loaded the client for us, all we have to do is return the client as a JSON object. If we wanted to, we could update the client secret and registration access token before returning the client's information, but that's left as an exercise for the reader.

```
app.get('/register/:clientId', authorizeConfigurationEndpointRequest,
function(req, res) {
   res.status(200).json(req.client);
   return;
});
```

Next we'll handle the update function. First check to make sure the client ID and client secret (if supplied) match what's already stored in the client.

```
if (req.body.client_id != req.client.client_id) {
   res.status(400).json({error: 'invalid_client_metadata'});
   return;
}

if (req.body.client_secret && req.body.client_secret !=
req.client.client_secret) {
   res.status(400).json({error: 'invalid_client_metadata'});
}
```

Then we need to validate the rest of the incoming client metadata. We're going to use the same client metadata validation function as in the registration step. This function will filter out any input fields that aren't supposed to be there, such as `registration_client_uri` and `registration_access_token`.

```
var reg = checkClientMetadata(req, res);
if (!reg) {
   return;
}
```

Finally, copy over the values from the requested object into our saved client and return it. Since we're using a simple in-memory storage mechanism, we don't have to copy the client back into the data store, but a database-backed system may have such requirements. The values in `reg` are internally consistent and will directly replace anything in `client`, and if they're omitted they will overwrite the values in `client`.

```
__.each(reg, function(value, key, list) {
   req.client[key] = reg[key];
});
```

[redacted]

Once that copy is completed, we can return the client object in the same manner as in the read function.

```
res.status(200).json(req.client);
return;
```

For the delete function, we have to remove the client from our data storage. We're going to do this using a couple of library functions from the Underscore.js library to help us out.

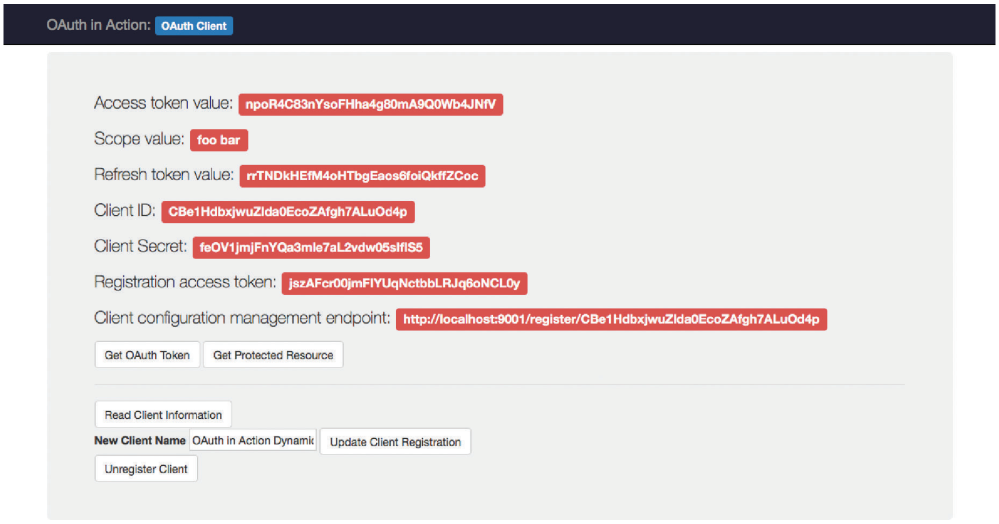
```
clients = __.reject(clients, __.matches({client_id: req.client.client_id}));
```

We're also going to do our due diligence as an authorization server and immediately revoke all outstanding tokens, whether they're access tokens or refresh tokens, that were issued to this client before we return.

```
nosql.remove(function(token) {
   if (token.client_id == req.client.client_id) {
         return true;
   }
}, function(err, count) {
   console.log("Removed %s clients", count);
});

res.status(204).end();
return;
```

With these small additions, the authorization server now supports the full dynamic client registration management protocol, giving dynamic clients the ability to manage their full lifecycle.

Now we're going to modify our client to call these functions, so edit `client.js`. Loading up the client and fetching a token displays an extra set of controls on the client's homepage (figure 12.4).



OAuth in Action: OAuth Client

Access token value: npoR4C83nYsoFHha4g80mA9Q0Wb4JNfV

Scope value: foo bar

Refresh token value: rrTNDkHEfM4oHTbgEaos6foiQkffZCoc

Client ID: CBe1HdbxjwuZlda0EcoZAfgh7ALuOd4p

Client Secret: feOV1jmjFnYQa3mle7aL2vdw05slflS5

Registration access token: jszAFcr00jmFlYUqNctbbLRJq6oNCL0y

Client configuration management endpoint: http://localhost:9001/register/CBe1HdbxjwuZlda0EcoZAfgh7ALuOd4p

Figure 12.4 The client homepage showing a dynamically registered client ID and controls to manage the registration

[redacted]

Let's wire up some functionality to those shiny new buttons. First, to read the client data, we're going to make a simple `GET` call to the client's configuration management endpoint and authenticate using the registration access token. We'll save the results of the call as our new `client` object, in case something changed, and display them using our protected resource viewer template to show the raw content coming back from the server.

```
app.get('/read_client', function(req, res) {

   var headers = {
        'Accept': 'application/json',
        'Authorization': 'Bearer ' + client.registration_access_token
   };

   var regRes = request('GET', client.registration_client_uri, {
        headers: headers
   });

   if (regRes.statusCode == 200) {
        client = JSON.parse(regRes.getBody());
        res.render('data', {resource: clien});
        return;
   } else {
        res.render('error', {error: 'Unable to read client ' +
   regRes.statusCode});
        return;
   }

});
```

Next we'll handle the form that allows us to update the client's display name. We need to make a clone of the client object and delete out the extraneous registration fields, as discussed previously, and replace the name. We'll send this new object to the client configuration endpoint in an HTTP PUT along with the registration access token. When we get a positive response from the server, we'll save that result as our new client object and go back to the index page.

```
app.post('/update_client', function(req, res) {

   var headers = {
        'Content-Type': 'application/json',
        'Accept': 'application/json',
        'Authorization': 'Bearer ' + client.registration_access_token
   };

   var reg = __.clone(client);
   delete reg['client_id_issued_at'];
   delete reg['client_secret_expires_at'];
   delete reg['registration_client_uri'];
   delete reg['registration_access_token'];

   reg.client_name = req.body.client_name;

   var regRes = request('PUT', client.registration_client_uri, {
```

[redacted]

```
          body: JSON.stringify(reg),
          headers: headers
     });

     if (regRes.statusCode == 200) {
          client = JSON.parse(regRes.getBody());
          res.render('index', {access_token: access_token, refresh_token:
     refresh_token, scope: scope, client: client});
          return;
     } else {
          res.render('error', {error: 'Unable to update client ' +
     regRes.statusCode});
          return;
     }

});
```

Finally, we'll handle deleting the client. This is a simple DELETE to the client configuration endpoint, once again including the registration access token for authorization. Whatever result we get back, we throw away our client information because, from our perspective, we did our best to unregister the client, whether the server was able to do so or not.

```
app.get('/unregister_client', function(req, res) {

     var headers = {
          'Authorization': 'Bearer ' + client.registration_access_token
     };

     var regRes = request('DELETE', client.registration_client_uri, {
          headers: headers
     });

     client = {};

     if (regRes.statusCode == 204) {
          res.render('index', {access_token: access_token, refresh_token:
     refresh_token, scope: scope, client: client});
          return;
     } else {
          res.render('error', {error: 'Unable to delete client ' + regRes.
     statusCode});
          return;
     }

});
```

With these in place, we've got a fully managed, dynamically registered OAuth client. Advanced client handling, including editing other fields, rotation of client secrets, and registration access tokens, is left as an exercise to the reader.

[redacted]

12.5 Summary

Dynamic client registration is a powerful extension to the OAuth protocol ecosystem.

- Clients can dynamically introduce themselves to authorization servers, but they still need a resource owner's authorization to access protected resources.
- Client IDs and client secrets are best issued by the authorization server that will accept them.
- Client metadata describes many attributes about the client and it can be included in a signed software statement.
- The dynamic client registration management protocol provides a full set of life-cycle management operations for dynamically registered clients over a RESTful API.

Now that you know how to introduce clients to authorization servers programmatically, let's take a look at one common application of OAuth: end-user authentication.

[redacted]